Relazione progetto C++ Aprile 2020

Nome: Benito Francesco Antonio

Cognome: Duretto

Matricola: 829795

Mail: b.duretto@campus.unimib.it

Introduzione

Il progetto consisteva nell'implementare un albero binario di ricerca che implementasse in primis i metodi fondamentali, quali costruttori e distruttore, e i metodi per aggiungere e cercare un nodo e quello di copiare una parte di albero. Inoltre il modo di confrontare due nodi doveva essere scelto dall'utente.

Tipi di dato nella BST

La BST è costituita da nodi, questi sono implementati come una struct privata della classe BST formalizzata come:

- Value di tipo T, ovvero il valore dato con un tipo definito nel main da inserire nella BST
- Left un puntatore al nodo sinistro
- Right un puntatore al nodo destro
- Parent un puntatore al nodo padre (ne discuterò nelle scelte implementative)

La BST ha, poi, una lunghezza che pian piano aumenta con l'aggiungere di nuovi nodi e una radice che punta ad un nodo.

Implementazione della BST

La BST è una classe templata che richiede un valore e due per il confronto (<>, ==) e ha 3 tipi di costruttori:

- Di default, che setta la radice a nullptr e la grandezza a 0
- Di copia, che, dato un altro albero come parametro, copia, se possibile, i nodi dell'albero copiandone anche la struttura. Usa copy_helper(metodo ricorsivo).
- Secondario templato, che permette di riempire l'albero dando una sequenza di dati

Ha un distruttore che elimina tutti i nodi creati e allocati in memoria.

Metodi della BST

Il metodo size ritorna l'effettiva grandezza dell'albero

Il metodo add inserisce nella BST un nodo con un determinato valore T, questo nodo è inserito seguendo le direttive di un funtore confronto dato dall'utente alla creazione dell'albero. Questo metodo non permette l'inserimento di dati duplicati: se un valore risulta essere già presente nella BST allora evita direttamente di inserirlo.

Il metodo find controlla se effettivamente è presente un determinato valore, usa find_helper per trovare l'esistenza di un nodo.

Il metodo subtree ritorna per valore un sotto albero con radice il valore passato. Se il valore passato è inesistente nella BST, ritorno l'eccezione bst_value_not_found. Questo metodo usa find_helper e copy_helper.

Il metodo privato clear lo uso principalmente nel distruttore e, poi, nei casi in cui debba svuotare il contenuto della bst per non avere dati inconsistenti.

Il metodo privato copy_helper, mi permette, in modo ricorsivo, di copiare i valori di un determinato albero ad una determinata posizione dettata dal nodo passato.

Il metodo privato find_helper mi permette di trovare un nodo all'interno di un albero e ritornare il puntatore. Mi è utile soprattutto per non duplicare codice in quanto lo uso in find e in subtree.

Ogni metodo pubblico della bst, escluso l'add, l'ho reso costante, perché non cambiano in nessun modo lo stato dell'albero e quindi chiamabili tranquillamente da un albero definito costante.

La BST ha un iteratore costante di tipo forward, che lancia l'eccezione bst_out_of_bound quando si prova ad incrementare l'iteratore quando il nodo puntato è nullo.

Il valore di begin dell'iteratore, lo inizializzo con il valore più a sinistra dell'albero perché seguo un ordine crescente, mentre il valore dell'end è nullo.

Per la stampa dell'albero nell'operator<< sfrutto gli iteratori. Le strutture dati inserite nell'albero devono avere la propria ridefinizione di <<.

Il metodo templato globale printIF stampa i valori dell'albero che rispettano un determinato predicato. Come per l'operator<<, sfrutto gli iteratori.

Main

Nel main richiamo delle funzioni di test che usano ogni metodo pubblico della BST, testando l'albero con vari tipi di valori: interi, stringhe, struct "point", BST di interi e BST di point.

Scelte implementative

- L'iterazione va in ordine da sinistra verso destra (crescente), per semplicità di comprensione nella stampa degli elementi e per un controllo 1 a 1 dei valori dell'albero.
- Inizialmente l'iteratore volevo crearlo sfruttando uno stack, ma francamente non mi sembrava per nulla elegante come soluzione. Quindi ho optato nell'inserire il parent nei nodi per avere un quadro più ampio. Leggendo sul libro di testo di algoritmi "Introduzione agli algoritmi e strutture dati – Cormen-McGraw-Hill", una BST ha proprio i nodi con il parent.
- Il costruttore di copia, per conseguenza alla prima scelta, non ho potuto farlo con gli iteratori, altrimenti creavo delle liste perdendo di fatto la loro utilità, e quindi ho optato per un metodo ricorsivo preorder con l'aiuto di copy_helper.
- Il costruttore secondario, che permette il riempimento con una sequenza generica di dati, l'ho implementato più per una questione pratica che altro. Mi serviva per avere ordinato il main nel riempimento di un albero, senza avere necessariamente bisogno di usare esplicitamente la funzione add, soprattutto quando istanzio una BST di BST e per creare un albero costante.